The Performance Cost of Preserving Data/Query Privacy Using Searchable Symmetric Encryption

The Performance Cost of Preserving Data/Query Privacy Using Searchable Symmetric Encryption

Shaun Mc Brearty[1], William Farrelly[1], Kevin Curran[2]

[1]Letterkenny Institute of Technology, Donegal, Ireland

[2]Ulster University, Derry, Northern Ireland
Email: kj.curran@ulster.ac.uk

Abstract

The benefits of Cloud computing include reduced costs, high reliability, as well as the immediate availability of additional computing resources as needed. Despite such advantages, Cloud Service Provider (CSP) consumers need to be aware that the Clouds poses its own set of unique risks that are not typically associated with storing and processing one's own data internally using privately owned infrastructure. New techniques such as Searchable Encryption are being deployed to enable data to be encrypted online. Despite being a relatively obscure form of Cryptography, Searchable Encryption is now at the point that it can be deployed and used within the Cloud. Searchable Encryption can allow CSP customers to store their data in encrypted form, while retaining the ability to search that data without disclosing the associated decryption key(s) to CSPs. Searchable Encryption is a diverse subject that exists in many forms. Searchable Symmetric Encryption (SSE) which has its roots in plaintext searching is one such form. Although symmetrically encrypted ciphertext cannot be searched in the same manner; nonetheless, many of the principles that apply to plaintext searching also apply to SSE. In its most basic form, SSE is nothing more than an Inverted Index – a mechanism that has been used in plaintext Information Retrieval (IR) for decades - that has been modified and adapted for use with ciphertext. We implement an SSE scheme and evaluate the efficiency of storing and retrieving data from the cloud.

The results showed that carrying out a task using SSE is directly proportional to the amount of information involved. In the case of constructing an IR Inverted Index, the results show that the time taken to generate an IR Inverted Index is directly proportional to the number of Terms contained in the underlying Document Collection. Converting the same IR Inverted Index to an SSE Inverted Index is directly proportional to the number of Postings contained within the IR Inverted Index, while the time taken to encrypt the underlying Document Collection is directly proportional to the number of Terms contained within the Document Collection. In relation to searching in SSE, the time taken to identify and decrypt the set of Postings associated with a given Lexicon Term is directly proportional to the number of Postings. We believe that SSE is efficient enough to be deployed in a Cloud environment especially when results only have to be returned to the user in small quantities. When applied to large Data Sets, SSE querying can become inefficient as its search time is directly proportional to the number of matching. SSE however is designed to achieve efficient search speeds whilst maintaining Data Privacy.

1. Introduction

The concept of Cloud computing is now an accepted philosophy for computing. As of 2014, 19% of all enterprises within the European Union utilise Cloud computing in some form or another, with industry forecasts indicating significant growth in the sector over the coming years ((Eurostat, 2014). The benefits of Cloud computing are significant: reduced costs, high reliability, as well as the immediate availability of additional computing resources as and when needed. Despite such advantages, Cloud Service Provider (CSP) consumers need to be aware that the Clouds poses its own set of unique risks that are not typically associated with storing and processing one's own data internally using privately owned infrastructure (Hashizume *et al.* 2013). Perhaps the most severe risk facing CSP consumers at present is the threat of data disclosure or data loss. Recent years have seen a number of such incidents occur, whereby organisations customer data – hosted on the Cloud - has been leaked online (for hacktivism or vandalism purposes) or stolen for criminal purposes. Cloud computing is made possible through the use of many technologies, including internet access, virtualisation and third party data centres. As a result, Cloud computing has a much broader attack surface than that associated with storing and processing data internally using privately owned infrastructure. The storing of consumer data online makes such information – potentially - accessible to anyone with a web browser, while the use of

virtualisation technology has the potential to allow CSP consumers to gain access to other CSP consumer's private data and/or applications (Hashizume *et al.*, 2013). In addition, the use of third party data centres poses a number of potential risks, including employees of the CSP (both current and former) gaining access to private consumer data (either physically or via software) (Hashizume *et al.* 2013, Nguyen *et al.* 2014). As a countermeasure to such attacks, various access controls are utilised: In the case of online access to the CSP, such access controls typically take the form of usernames and passwords; In the case of virtualisation, such access controls typically take the form of logical data separation; and in the case of third party data centres, such access controls typically take the form of physical access controls (*For Example: Locks, Keypads*) (as well as software based access control) that prevent unauthorised CSP personnel from gaining access to user data (Hashizume *et al.* 2013). In principle, all of the aforementioned access controls are sound; however in practice, such controls have been circumvented.

In the event that any of the aforementioned access controls are compromised maliciously, the chances of a data breach occurring are high. Should a data breach occur and the associated data is retrieved in encrypted form, the data is essentially useless to an attacker (unless the encryption algorithm utilised is weak and/or the attacker has some foreknowledge of the associated decryption key) (Hashizume *et al.* 2013); however, in the event that a data breach occurs and the associated data is retrieved in plaintext form, an organisations worst nightmare has become a reality. What follows is typically a slew of press releases, negative publicity, damaged business reputations, and fines under various data protection laws (ICO, 2015, Levick, 2015). To reduce the impact of potential data breaches (and to provide privacy for CSP consumer data) CSPs typically employ the use of cryptography. In a Cloud environment, cryptography is typically utilised for two purposes: security while data is at rest; and security while data is in transit. Unfortunately the Cloud cannot guarantee the security of data during processing as the current limitations of cryptography prevent data from being processed in encrypted form. Given the fact that data is processed in unencrypted form, it is quite common for attackers to target data in use, rather than targeting data which is encrypted during storage and transit (Hashizume *et al.*, 2013). An entity wishing to store its data within the Cloud must choose one of the following options:

1. Store Data in Encrypted Form (Two Options Exist)
    A. Disclose Decryption Key(s) to Cloud Service Provider (CSP) OR
    B. Keep Decryption Key(s) Private
2. Store Data in Unencrypted Form

Option 1A requires encrypted data owners to disclose their decryption key(s) to CSPs. This is due to the fact that data cannot be searched or operated on while in encrypted form. In order to provide CSP customers with such functionality, CSPs require access to the necessary decryption key(s). Option 1B (Keeping Decryption Key(s) Private) represents the most secure sub-option; however, as previously mentioned, CSP customers lose the ability to search or operate on their data while it is in encrypted form. In order to utilise such functionality using Option 1B, CSP customers must download their data, decrypt it, and only then can it be searched and/or operated on. While this approach may be fine for small amounts of data, it becomes increasingly inefficient and unwieldy as the amount of data increases. In addition, should any changes be made to the data after it has been downloaded; the customer must then re-encrypt and re-uploaded the entire dataset to the Cloud. Option 2 avoids the use of encryption for data security. Rather than relying on cryptography for data security; *that is, the traditional approach to data security*, this approach utilises the aforementioned approach of logically separating data (Mather *et al.* 2009). Evidently, none of the options available at present provide an adequate balance of data security and functionality. Option 1A and Option 2 offer full functionality at the expense of data security, while Option 1B provides data security at the expense of any and all functionality. The ideal solution to achieving an optimal balance of data security and functionality within the Cloud involves the CSP having the ability to search and operate on data while it is in encrypted form – without having any knowledge of the associated decryption key(s), or the associated plaintext(s) (Mather *et al.* 2009).

Two forms of encryption do in fact exist at present that make the above a reality. The first, known as Fully-Homomorphic Encryption (FHE) allows data to be *operated on* while in encrypted form (Gentry 2009). The second, known as Searchable Encryption, allows for data to be *searched* while in encrypted (Song *et al.* 2000). While being impressive in terms of its functionality and capabilities, FHE remains extremely slow when implemented in software (Gentry *et al.* 2015). As such, its mass deployment and usage within the Cloud appears to be some way off (Wang et al., 2015).

Searchable Encryption on the other hand has been shown to be sufficiently efficient on the few occasions that it has been implemented in software (Uchide & Kunihiro, 2016). Despite being a relatively obscure form of Cryptography, Searchable Encryption is now at the point that it can be deployed and used within the Cloud (Kamara *et al.* 2012, Cash *et al.* 2013). Used in the Cloud, Searchable Encryption has the ability to allow CSP customers to store their data in encrypted form, while retaining the ability to search that data without disclosing the associated decryption key(s) to CSPs (Song *et al.* 2000), *that is, without compromising data security on the*

Server. Searchable Encryption is a diverse subject that exists in many forms. While there are several methods of carrying out Searchable Encryption, two general techniques dominate the literature: Searchable Symmetric Encryption (SSE); *that is, Searchable Encryption using Symmetric Key Cryptography* and Public Key Encryption with Keyword Search (PEKS); *that is, Searchable Encryption using Public Key Cryptography (Curtmola et al. 2006, Bosch et al. 2014)*. Neither SSE nor PEKS natively supports Searchable Encryption as it was originally envisioned by Song *et al.* (2000). Instead, the literature has focussed on adapting various forms of Indexes; *that is, Data Structures that support efficient searching by pre-computing and mapping Search Terms to the Documents they occur in (and vice versa)*, for use with Information Retrieval (IR) over encrypted Documents (Bosch *et al.* 2014).

Forward Indexes and Inverted Indexes store the exact same information, each Index is optimised for different forms of searching. In the case of the Forward Index, it is optimised for searching *specific Documents* for the presence of Search Strings (Luenberger 2006, p.285), while the Inverted Index is optimised for searching *an entire Document Collection* for the presence of Search Strings (Luenberger 2006; Manning *et al.* 2008). Early work on the topic of Searchable Encryption focussed on the use of Forward Indexes almost exclusively (Goh 2003, Chang and Mitzenmacher 2005); however, subsequent work on the topic has focussed on the use of Inverted Indexes (due to its ability to efficiently search an entire Document Collection, as opposed to specific Documents) (Curtmola *et al.*, 2006; Kamara *et al.*, 2012; Cash *et al.*, 2013). Aside from SSE and PEKS, two other forms of encryption exist at present that support Searchable Encryption: Fully-Homomorphic Encryption (FHE) (Gentry 2009) and Oblivious RAM (ORAM) (Goldreich and Ostrovsky 1992).

Fully-Homomorphic Encryption supports computations over data in encrypted form, including Searchable Encryption as it was originally envisioned by Song et al (2000); nonetheless efficient Fully-Homomorphic Encryption remains someway off (Gentry *et al.* 2015). Used in isolation, ORAM does not support Searchable Encryption. Essentially, ORAM is a Client-Server communication protocol designed to obfuscate memory access patterns on the Server side of a given transaction. In its simplest form, ORAM consists of two operations: The Client storing data on the Server; *that is, writing*, and the Client retrieving Data from the Server; *that is, reading*. In an effort to obfuscate memory access patterns on the Server, each Write operation is also accompanied by an associated Read operation, and each Read operation is accompanies by an associated Write operation. In addition, each Read/Write operation accesses numerous memory locations on the Server instead of just a single memory location (in an effort to further obfuscate memory access patterns; *that is, false positives*) (Goldreich and Ostrovsky 1992). In the context of Searchable Encryption, ORAM is typically combined with SSE and PEKS Searchable Encryption schemes to improve their security. SSE and PEKS Searchable Encryption schemes Leak Information to the Server a number of ways. By combining such schemes with ORAM, such Information Leakage can be eradicated; nonetheless, the search efficiency of schemes utilising ORAM is severely hindered due to the amount of work involved in obfuscating memory access patterns using ORAM (Stefanov *et al.* 2013). In relation to search efficiency, both SSE and PEKS achieve optimal search time when used in conjunction with an Inverted Index; *that is, search time is linear in the number of Documents matching the Search String*; however in terms of security, SSE is vastly superior to PEKS (Bosch *et al.* 2014). Given that PEKS is a form of Public Key encryption, an adversary can easily mount an attack on such a Searchable Encryption scheme given the associated Public Key and a dictionary of chosen Terms. In the case of SSE, all associated keys are kept private (Curtmola *et al.* 2006).

SSE represents one of the few forms of Searchable Encryption that is achievable using established standardised encryption algorithms (Cash et al., 2015). Alternative forms of Searchable Encryption require the use of non-standardised, special purpose encryption algorithms (Gentry 2009). SSE is considered one of the least secure forms of Searchable Encryption (see figure 1) primarily due to Information Leakage. Solutions exist to eradicate and obfuscate all forms of Information Leakage in SSE; however existing solutions have a significant effect on the search efficiency of SSE (Stefanov *et al.* 2013). Evidently, the challenge for researchers is to improve the security of SSE while maintaining its superior search efficiency (He et al., 2016).

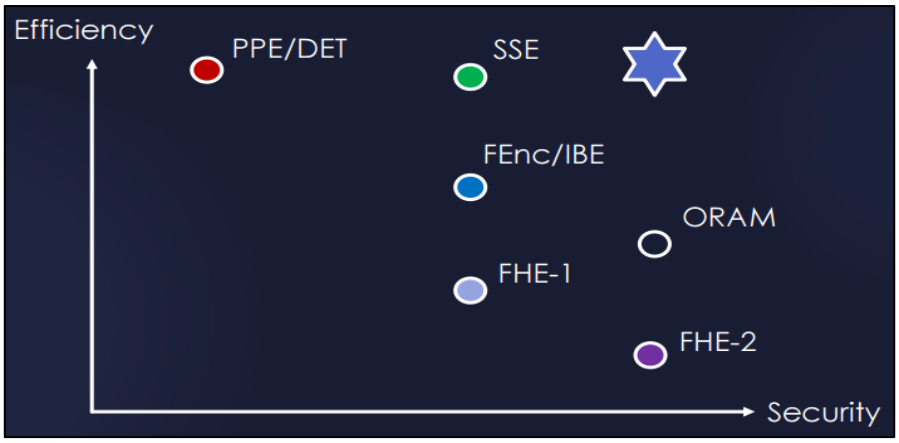

Figure 1: Efficiency Vs. Security Trade-off For Various Searchable Encryption Schemes (Kamara 2013)

Figure 1 lists all known solutions to the problem of searching on encrypted data; *that is, symmetrically encrypted data, as well as public key encrypted data*. The y-axis of figure 1 lists all Searchable Encryption solutions with respect to their efficiency, while the x-axis lists all solutions with respect to security. As regards efficiency, the SSE literature defines efficiency as the time-complexity associated with finding a given Encrypted Search String (ESS) within a body of encrypted data (expressed in Big O Notation). In terms of security, the SSE literature defines security as the amount of Information Leakage associated with using a given Searchable Encryption scheme; *that is, what the Server learns (or can deduce) about the ciphertext by searching over it* (expressed in Terms of the numerous categories of Information Leakage) (Kamara 2013).

2. Searchable Encryption

Song *et al.* (2000) first proposed the concept of Searchable Encryption using an example whereby the content of symmetrically encrypted Documents were sequentially searched; *that is, character-by-character, word-by-word*, for the presence of a user specified Search String. Prior to the Search taking place, the Search String specified by the user was first encrypted using the same key used to encrypt the Documents being searched, with the resulting value – referred to as the Encrypted Search String (ESS) – being the value Searched for within the encrypted Documents. Those encrypted Documents deemed to contain the ESS were then returned to the user as part of the subsequent Search Results (see Figure 2).

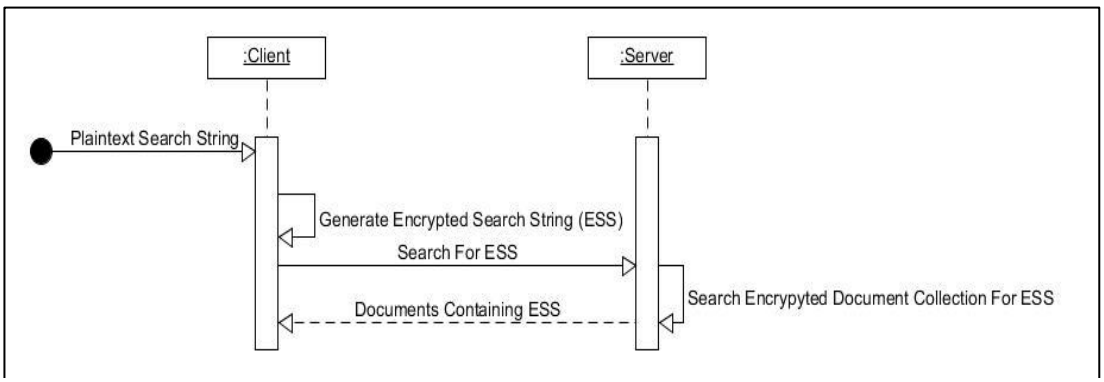

Figure 2: Original Description of Searchable Encryption

While this explanation successfully communicated the basic premise of Searchable Encryption – in a manner relatively similar to plaintext Information Retrieval (IR); *that is, plaintext searching* - it nonetheless ignored the fact that modern symmetric ciphers do not support Searchable Encryption as described by Song *et al.* (2000). Specifically, modern symmetric ciphers implement Shannon's Confusion and Diffusion principles (through the use of Substitution-Permutation networks) to counter cryptanalysis (Stallings 2014, pp.66-67). As a consequence, Searchable Encryption - as described by Song *et al.* (2000) is not feasible as the data must be encrypted using Electronic Code Book mode (the use of ECB mode is highly discouraged due to its susceptibility to cryptanalysis) and that the author of the Document being searched limit the maximum length of plaintext Terms in the Document to the Block Size of the associated cipher (8 characters in the case of DES, 16 characters in the case of AES). In addition, the scenario also requires that the author of the Documents ensure that only a single Term is contained within each ciphertext Block.

Searchable Encryption operates on the assumption that a given Term - whether in plaintext form or encrypted form - is located in the same position in both the plaintext version of the Document and the encrypted version of the same Document. *For Example: Given a plaintext Document beginning with the Term 'The', the description provided by Song et al. (2000) assumes that the first three characters of both the plaintext version of the Document and the encrypted version of the Document correspond to the Term 'The'.* Essentially this description assumes that symmetric ciphers encrypt data one character at a time, when in reality, this is not the case. Modern symmetric ciphers encrypt data in blocks of a fixed size, rather than character by character (Stallings 2014). The effect of using such ciphers is that the ciphertext associated with a given plaintext Term is spread across the entire ciphertext block, rather than appearing in the same position as the plaintext Term; thus preventing traditional Sequential Searching (Stallings 2014). In addition, modern symmetric ciphers typically operate using advanced block cipher modes (another mechanism to counter cryptanalysis) which 'chain' the ciphertext of previously encrypted blocks to the current plaintext block (by means of a bitwise XOR operation) (Stallings 2014); thus further complicating the problem of searching ciphertext for the presence of an encrypted version of a plaintext Search String. Recognising the inherent difficulty in achieving Searchable Encryption as originally described by Song *et al.* (2000), subsequent work in the area focussed on developing solutions to the problem as originally conceived; albeit without actually using Sequential Searching (Goh 2003). Specifically, researchers focussed on adapting the Inverted Index – a mechanism that has been used in plaintext Information Retrieval for decades – for use in Searchable Encryption (Curtmola *et al.* 2006). In its most basic form, an Inverted Index is a Data Structure that maps Terms to the Document(s) they occur in; therefore eradicating the need to Sequentially Search Documents (Luenberger, 2006; Manning *et al,.* 2008). When adapted for use with an encrypted Document Collection, the resulting Inverted Index is titled Searchable Symmetric Encryption (SSE) (Curtmola *et al.,* 2006)- the topic of focus for this research.

2.1 Information Retrieval and the Inverted Index

Unlike searching a Collection of encrypted Documents, searching a Collection of plaintext Documents for the presence of a user specified Search String is a trivial process. The most basic method of doing so, known as Sequential Searching, involves examining each Document within a Collection on a Term by Term basis. As each Term within the Document being examined is encountered, the Term in question is simply compared to the user specified Search String for equality (assuming the Search String in question consists of a single Term). In the event that a Document Term matches the user specified Search String, the associated Document is then returned to the user as part of the ensuing Search Results (Manning *et al,.* 2008). While Sequential Searching functions effectively, its search efficiency is poor: Sequential Searching suffers from the fact that each Document in the Collection must be examined; therefore making its search time linear in the number of Documents contained within the Collection. As such, the time taken to search the Collection increases as the number of Documents in the Collection expands. The poor performance of Sequential Searching can be directly attributed to the fact that the set of Terms contained within each Document must be determined at run time; *that is, while the Search is being conducted*. In addition, the set of Documents that a Search String occurs in must also be determined at run-time; hence why each Document within the Collection must be examined (Manning *et al.* 2008). In an effort to expedite the process of plaintext Information Retrieval (IR), the Inverted Index was developed. Just like a Database Index is designed to speed up data retrieval without searching each row of a Database Table, the Inverted Index is designed to speed up IR without having to search each Document within a Collection. In its most basic form, an Inverted Index is a Data Structure that maps each Term within a Collection to the Document(s) it occurs in (Luenberger, 2006). The Inverted Index attempts to overcome the shortcomings of Sequential Searching by pre-computing the list of all Terms contained within a Document Collection, as well as each pre-computing what Document(s) each Term occurs in; *that is, in advance of a search occurring*. The purpose of pre-computing this list of Terms – commonly referred to as the *Lexicon* of the Collection – is that the list of Terms is searched for the presence of the user specified Search String, instead of the Document Collection; thus making the search time linear in the number of Terms contained within the Collection. For improved search efficiency, it is common for the Lexicon to be stored using Data Structures that expedite searching, such as Hash Tables (*O(1) Search Complexity*), Binary Search Trees (*O(Log N) Search Complexity*), B-Trees (*O(Log N) Search Complexity*) or Word Tries (*O(Log N) Search Complexity*).

2.1.1 Inverted Index Construction

Construction of an Inverted Index first requires a Document Collection from which the Inverted Index will be built; *that is, the Document Collection to be searched*. Inverted Index construction begins with each Document within the Collection being sequentially scanned by the Server, and a note being made of each Term that occurs within each Document. In SSE, the Client is responsible for constructing the Inverted Index; not the Server as is the case with plaintext Information Retrieval (IR). This process is typically referred to as *Document Tokenisation* (Manning *et al.* 2008). Each and every Term encountered during Document Tokenisation is added to a list known as the *Lexicon* (see Figure 3). Essentially, the Lexicon is the list of all Terms that occur in a given Document Collection. In the event of the same Term occurring multiple times in a single Document, or

the same Term occurring in multiple Documents within the Collection– both of which are inevitable - the Term in question appears only once in the Lexicon; therefore, each Term contained within the Lexicon is unique (Luenberger 2006).

Lexicon
a
at
attack
be
bed
before
bend
chair

Figure 3: Sample Lexicon

Lexicon	Postings (Document IDs)
a	1, 2, 3, 4, 5
at	1, 4, 5
attack	1, 2, 5
be	3, 4, 5
bed	4, 5
before	1, 4
bend	1, 6
chair	3

Figure 4: Sample Lexicon (Including Postings/Posting Lists)

Throughout the process of Document Tokenisation, each and every Document that a given Term occurs in is also noted; *that is, the Document ID is noted* (see Figure 4). The noting of a given Term occurring in a given Document is referred to as a *Posting*, while the list of all Documents; *that is, Document ID's*, where a given Term occurs is referred to as a *Posting List.*

Lexicon	Doc 1	Doc 2	Doc 3	.....	Doc N
Term1	1	0	1	.....	0
Term 2	0	1	1	.....	1
Term 3	1	1	1	.....	1
⋮					⋮
Term M	1	0	0	.....	1

Figure 5: Tabular Visualisation of an Inverted Index (Luenberger 2006)

Figure 5 depicts a simple tabular visualisation of an Inverted Index. The Lexicon for the Document Collection is listed in the left most column of the table, while the list of Documents within the Collection; *that is, Documents IDs* is listed along the top row of the table. The intersection of each row and column contains a value denoting whether or not the Term associated with the row in question occurs within the Document associated with the column in question, with '1' denoting the occurrence of the Term within the Document; *that is, a Posting*, and '0' denoting the absence of the Term from the Document. Regarding implementation details, an Inverted Index can be implemented in a number of ways:

- As mentioned previously, the Lexicon can be implemented and stored using a number of different Data Structures. This research assumes that the Data Structure used is a Hash Table (due to its efficiency; *that is, O(1)*).
- Due to their list-like nature, Posting Lists are typically implemented using a Linked List Data Structure. Alternative Data Structures, such as Arrays, can be used; however the dynamic nature of Posting Lists often makes the Linked List Data Structure the preferred choice.
- In relation to Document storage, Documents can be stored in a number of ways. Primarily, Documents are either stored using the native file system of the Server they are stored on, or alternatively, as Rows within a Database Table (with their designated Document ID acting as their Primary Key value).

In term of memory management, the Lexicon of an Inverted Index is typically loaded into Random Access Memory (RAM) at all times. Given that the Lexicon contains the information to be searched whenever an Inverted Index is queried; it is therefore common that a significant amount of RAM be allocated to same. Regarding memory management for Posting Lists and Documents, both sets of information are typically stored in secondary memory. This is due to the fact that both Posting Lists and Documents are only ever retrieved whenever their associated Terms are searched for. For improved performance, it is common for the first Link of a Postings List Linked List to be stored alongside its associated Term in the Lexicon Data Structure; *that is, in RAM*. This is due to the fact that the first Link in a Linked List is required to access all subsequent Links in the Linked List; *that is, all subsequent Postings* (stored in secondary memory). Performing a Query against the Inverted Index structure is relatively simple. Given a Search String, the Lexicon Data Structure is examined to

determine the presence or absence of the Search String within the Lexicon. In the event that the Search String is present in the Lexicon, the first Posting associated with the matching Term is retrieved from RAM. In turn, this Link is then used to retrieve all subsequent Links in the Linked List (stored in secondary memory); thus retrieving all Postings for the Search String in question. Once the Posting List has been retrieved in full, the associated Document IDs are then used to retrieve the actual Documents – from secondary memory – that contain the Search String. Once all Documents are retrieved, they are then forwarded to the Client; *that is, Search Results*. In the event that a Search String is not present in the Lexicon, this denotes that the Search String in question is not present in any Documents contained within the Collection.

2.2 Searchable Symmetric Encryption (SSE)

To ensure clarity, we refer to the Inverted Index structure used in plaintext Information Retrieval (IR) as the IR Inverted Index, while its SSE counterpart is referred to as the SSE Inverted Index. As the name suggests, the SSE Inverted Index borrows heavily from the IR Inverted Index. All information presented previously in relation to the IR Inverted Index remains true for the SSE Inverted Index; however the reader should be aware that SSE and the SSE Inverted Index differ from IR and the IR Inverted Index in the following ways:

The topic of Information Leakage forms an Integral part of SSE. When the idea of Searchable Encryption was first proposed, one of its founding principles was the assumption that the Server storing the encrypted Document Collection is an adversary that is actively working on subverting the security of the Document Collection it possesses (with the ultimate goal of gaining access to the Document Collection in plaintext form) (Song *et al.* 2000). As such, the SSE Inverted Index is constructed and operates in a manner that takes significant steps to reduce the Leakage of potentially useful Information to the Server. In practice, this involves the use of encryption for the Document Collection, the Lexicon, Posting Lists and Search Strings; as well as the use of Data Structures that hinder the Servers efforts in achieving its malicious goals (Goh 2003, Chang and Mitzenmacher 2005, Curtmola *et al.* 2006).

Responsibility for creating the SSE Inverted Index is offloaded to the Client. In order for the Server to construct the SSE Inverted Index, decryption keys must be disclosed to the Server (as mentioned previously, this is undesirable from a data security perspective). Rather than reveal sensitive information to the Server, SSE delegates responsibility of constructing the SSE Inverted Index to the Client. Given that the Client is responsible for constructing the SSE Inverted Index, it is therefore expected that the Client forwards the SSE Inverted Index to the Server along with the encrypted Document Collection whenever the latter is forwarded to the Server for storage (Goh 2003).

2.2.1 Information Leakage

A significant portion of the Searchable Encryption literature has focussed on determining what Information Leakage results from a) The Server being in possession of the encrypted Document Collection, and b) The Server carrying out searches on same; *that is, a Client ordering the Server to perform a Search, or the Server itself carrying out searches covertly*. The purpose of studying such Information Leakage is to determine whether or not any and all Information Leaked by various Searchable Encryption schemes is useful to the Server in terms of achieving its malicious goal(s). Ideally, no Information Leakage should occur as a result of utilising Searchable Encryption; however, like all ideal scenarios, realising it is not without its challenges. The two most secure forms of Searchable Encryption at present; *that is, Oblivious RAM (RAM) and Fully Homomorphic Encryption-2 (FHE-2)* achieve zero Information Leakage; however both do so at the expense of efficiency. In both solutions, this poor efficiency can be directly attributed to the Information Leakage countermeasures utilised (Stefanov *et al.* 2013). In an effort to improve the overall efficiency of Searchable Encryption, several researchers have examined the prospect of relaxing the zero-tolerance approach to Information Leakage in Searchable Encryption (Goh, 2003, Chang and Mitzenmacher, 2005). Specifically, researchers have attempted to determine what Information Leakage is acceptable in Searchable Encryption (sometimes referred to as Trivial Information Leakage); *that is, Information that in no ways aids the Server in achieving its goal of subverting the encrypted Document Collection*. Evidently, the goal of this Research was to identify which Information Leakage countermeasures are absolutely necessary in Searchable Encryption; therefore allowing researchers to focus on creating search efficient schemes that conform to this baseline measure of Information Leakage (at a minimum).

In the case of Searchable Symmetric Encryption (SSE), it is the SSE Inverted Index that is searched by the Server, instead of the encrypted Document Collection. As such, the SSE literature instead focuses on determining what Information Leakage results from the Server being in possession of the SSE Inverted Index, as

well as what Information Leakage results from the Client (or the Server itself) querying same. In terms of Information Leakage in SSE, such Information Leakage is typically broken into three categories: Storage Leakage; *that is, what the Server can learn (or deduce) from the SSE Inverted Index by simply storing it (that is, without the SSE Inverted Index actually being queried)*, Query Leakage; *that is, what the Server can learn (or deduce) from the SSE Inverted Index by querying it itself (covertly) , or observing the SSE Inverted Index being queried by Client(s)*, and Update Leakage; *that is, what the Server can learn (or deduce) whenever the SSE Inverted Index is updated (such as when Documents within the Collection are edited/deleted, or when new Documents are added to the Collection)* (Curtmola *et al.* 2006, Chase and Kamara 2010, Kamara *et al.* 2012, Kamara 2013).

We have attempted to produce an all-encompassing list of Information that can potentially be Leaked by SSE, as well as highlighting what Information Leakage is classified as Trivial and Non-Trivial by the Searchable Encryption literature (Curtmola *et al.* 2006, Cash *et al.* 2013). Figure 6 presents potential Storage Leakage by an SSE scheme (including an indication of whether or not such Information is classified as Trivial Information Leakage or Non-Trivial Information Leakage), while Figure 7 presents potential Query Leakage by an SSE scheme. Note that potential Storage Leakage is presented in terms of the three inter-related data sets that make up an SSE Inverted Index; *that is, the Lexicon, Postings and the associated encrypted Document Collection*, while potential Query Leakage is presented in terms of the Search Pattern; *that is, the Encrypted Search String (ESS) received by the Server (and whether or not the ESS was utilised before)*, and the Access Pattern; *that is, those encrypted Document(s) deemed to contain the ESS, their associated memory location(s) and their Document ID(s)*. In the description of SSE that follows, all Information classified as Non-Trivial in Figure 6 (Potential Storage Leakage) and Figure 7 (Potential Query Leakage) is Leaked to the Server.

Lexicon	**Trivial**	**Non-Trivial**
Lexicon Terms In Plaintext Form		X
Lexicon Terms In Ciphertext Form	X	
Total Number of Lexicon Terms In Inverted Index	X	
Length of Individual Lexicon Terms In Plaintext Form		X
Length of Individual Lexicon Terms In Ciphertext Form	X	
Postings		
Number of Postings Per Lexicon Term, *i.e. Term-Document Frequency (TDF)*		X
Total Number of Postings In Inverted Index	X	
Document IDs In Plaintext Form		X
Document IDs In Ciphertext Form	X	
Document Collection		
Individual Documents In Plaintext Form		X
Individual Documents In Ciphertext Form	X	
Size of Individual Documents	X	
Total Number of Documents In Collection	X	

Figure 6: Potential Storage Leakage in SSE (Including Trivial Leakage and Non-Trivial Leakage).

Search Pattern	**Trivial**	**Non-Trivial**
Search String In Plaintext Form		X
Length Of Search String In Plaintext Form		X
Search String In Ciphertext Form	X	
Length Of Search String In Ciphertext Form	X	
Whether Or Not A Given Search String Has Been Searched For Previously	X	
Access Pattern		
Matching Document IDs In Ciphertext Form	X	
Matching Document IDs In Plaintext Form	X	
Memory Locations Of Matching Documents	X	
Matching Documents In Plaintext Form		X
Matching Documents In Ciphertext Form	X	
Search Pattern/Access Pattern Combined		
Time Taken For Query To Execute	X	
Number of Rounds of Interaction Between Client and Server	X	
Number of Documents Containing Search String, *i.e. TDF*	X	

Figure 7: Potential Query Leakage in SSE (Including Trivial Leakage and Non-Trivial Leakage).

From examining Figure 6 and Figure 7, it is evident that Leaking Information to the Server in ciphertext form is considered Trivial Information Leakage by the SSE Literature – irrespective of whether such ciphertext is

leaked to the Server as part of Setup Leakage or Query Leakage. As regards plaintext Information Leakage, such Leakage is *generally* considered Non-Trivial Information Leakage by the SSE Literature; however, a notable exception to this rule is Document IDs. From Figure 6 (Postings Section), it is noticeable that the Leakage of Document IDs in plaintext form is considered Non-Trivial Information Leakage in the case of Setup Leakage; while the exact same Information is classified as Trivial Information Leakage in the case of Query Leakage (Figure 7 – Access Pattern).

In addition to considering Information Leakage in the context of plaintext Information and ciphertext Information, the literature also considers Information Leakage from a statistical point of view; *that is, those statistics that be derived from Information, irrespective of whether the underlying Information is in plaintext form or ciphertext form*. Generally, the SSE literature classifies statistical Information Leakage as Trivial Information Leakage; however one exception does exist. The statistic in question – known as Term-Document Frequency (TDF)*; that is, the number of Documents containing a given Term* - is classified as Non-Trivial Information Leakage in the case of Setup Leakage (see Postings in Figure 6); and Trivial Information Leakage in the case of Query Leakage (see Access Pattern in Figure 7) (much like Document IDs as discussed previously). Admittedly, the decision to label certain Information Leakage as Non-Trivial for Setup Leakage and the decision to label the exact same Information Leakage as Trivial for Query Leakage appears bewildering. Nonetheless, this can be explained by the conservative approach to Information Leakage taken by researchers in the area. Generally, where certain Information Leakage is considered unavoidable (and the Information in question is classified as Trivial Information Leakage), researchers take the approach of allowing such information to be Leaked; however, rather than Leak such Information immediately; *that is, Storage Leakage*, researchers will typically guard such Information up to the point where its Leakage is absolutely necessary and therefore unavoidable (otherwise known as Controlled Disclosure); *that is, Query Leakage* (Curtmola *et al.* 2006, Chase and Kamara 2010, Cash *et al.* 2013).

2.2.2 SSE Inverted Index Construction

The steps involved in constructing an SSE Inverted Index are exactly the same as those involved in constructing an IR Inverted Index, albeit the Client has responsibility for generating the SSE Inverted Index, and various forms of encryption are applied to each dataset after they have been compiled; *that is, the Document Collection, the Lexicon and the Postings List (Goh 2003)*. In addition to the use of encryption, a different Data Structure – namely, an Array - is utilised to store Postings instead of a Linked List (as is used in the IR Inverted Index) (Curtmola *et al.* 2006). Rather than storing Lexicon Terms in plaintext form, SSE requires that a keyed-hash of each Term be stored instead (Chase and Kamara 2010, Kamara *et al.* 2012). The use of a keyed hash function for this purpose - instead of traditional reversible encryption - may seem curious at first; however researchers have successfully argued that the Lexicon's sole purpose within the Inverted Index is to provide the Client with the ability to carry out searches and nothing more. Given that the Lexicon is unlikely to be downloaded to the Client (and is therefore unlikely to be decrypted - unlike the actual Documents), the use of reversible encryption for encrypting Lexicon Terms has largely been abandoned. Aside from the aforementioned reasons, the use of a keyed hash function for this purpose has a number of advantages in terms of reduced Information Leakage and improved data security, including the following (Stallings, 2014):

- First and foremost, the use of a hash function (keyed or non-keyed) ensures that all encrypted Lexicon Terms within the SSE Inverted Index are of equal length (*a hash function produces a Hexadecimal String of fixed length*); therefore masking the length of all underlying plaintext Lexicon Terms.
- Secondly, the use of a hash function (again, keyed or non-keyed) ensures that an adversary has no means of decrypting the encrypted Lexicon Term back to its plaintext form.
- Thirdly, ensuring that a keyed hash function is used – instead of a traditional non-keyed hash function – protects SSE from Rainbow Table Attacks; *that is, pre-computed Hash Values of common Dictionary Words*.

As regards keyed-hash algorithms, the SSE literature states that any standardised secure algorithm can be utilised for Lexicon Encryption (*For Example: HMAC-MD5, HMAC-SHA256*).

2.2.2.1 Postings List

The use of Linked Lists for Posting List storage is abandoned in SSE due to Setup Leakage resulting from their modus operandi; *that is, sequential memory access*, with Arrays being preferred instead (Curtmola *et al.* 2006). Specifically, given the first Link in a Linked List, it is a trivial process to examine all subsequent Links due to the fact that each Link in a Linked List contains a pointer to the next Link (see Figure 8). Given that each Term in an IR Inverted Index has its own dedicated Linked List to store Postings; it is therefore a trivial process to derive

the Term-Document Frequency (TDF) for each Term in the Lexicon in advance of the associated Term being searched for.

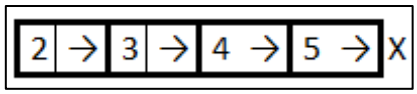

Figure 8: Linked List Data Structure (→ Denotes A Pointer to the Next Link in the Linked List).

Rather than using one Array for each Term in the Lexicon (doing so would also result in TDF Storage Leakage; *that is, the size of the Array would be equivalent to the TDF*), SSE utilises a single one dimensional Array to store all Postings for all Terms (see Figure 9). Utilising this approach, Setup Leakage amounts to the total number of Postings for the entire Lexicon*; that is, trivial Leakage*.

Array Index	1	2	3	4	5	6
Doc ID	4	7	1	9	8	7

Figure 9: Postings Stored In an Array.

Given that all Postings are now stored in a single one dimensional Array, some mechanism to keep track of what Postings belong to what Terms is therefore required. The solution to this problem is relatively similar to a Linked List, albeit the solution involved does not utilise pointers (as is the case with Linked Lists). In order to keep track of what Postings are associated with a given Term, SSE requires that the Document ID of the first Posting associated with a given Term is stored alongside the keyed-hash of the Term in the Lexicon Hash Table (in RAM) (*For Example: Doc ID 1*). Alongside this Document ID (in the Lexicon Hash Table) is an Array Index denoting the location of the second Posting associated with the Term (see Figure 10) (*For Example: 94*). At the Array Index in question is the Document ID of the 2nd Posting, as well as the Array Index denoting the location of the third Posting (*For Example: 79*) (see Figure 11).

Term (HEX)	1st Posting (Doc ID)	Location of 2nd Posting
A16GDB563E	1	94

Figure 10: SSE Lexicon Node.

2st Posting (Doc ID)	Location of 3rd Posting
2	79

Figure 11: Postings Array Node.

Rather than storing all Postings sequentially within the Array, SSE requires that all Postings be shuffled to random locations within the Postings Array. As such, the second Posting for a Term may be located at Array Index 1000, while the third Posting may be located at Array Index 1. Despite utilising Arrays and arranging Postings in non-sequential order, the fact remains that the Information stored at each Index of the Postings Array is in plaintext form. As such, it is still a trivial process for the Server to calculate the TDF for each Term in the Lexicon in advance of the Term being searched for (as was the case previously with Linked Lists). As a solution to this problem, SSE requires that each Document ID within the Postings Array be encrypted, as well as each 'Next Posting Location'; therefore preventing the Server from deducing this Information by merely being in possession of the SSE Inverted Index; *that is, Storage Leakage*. Rather than encrypting all Postings using the same key, SSE requires that each Posting be encrypted using a different key. In an effort to reduce the number of keys the Client has to remember in order to utilise SSE, the literature recommends the following guidelines for encrypting the Posting Array:

1. The encryption/decryption key for the first Posting associated with each Term should be derived by passing its associated plaintext Term through a keyed hash function (the key utilised within the keyed hash function is a master key known only to the Client).
2. All subsequent Postings in the Array; *that is, 2nd Posting, 3rd Posting, 4th Posting, etc.*, are to be encrypted/decrypted using randomly generated encryption keys.
3. The key required to decrypt a given Posting (with the exception of the first posting) is to be stored in the previous Posting associated with the Term in question (see Figure 12).

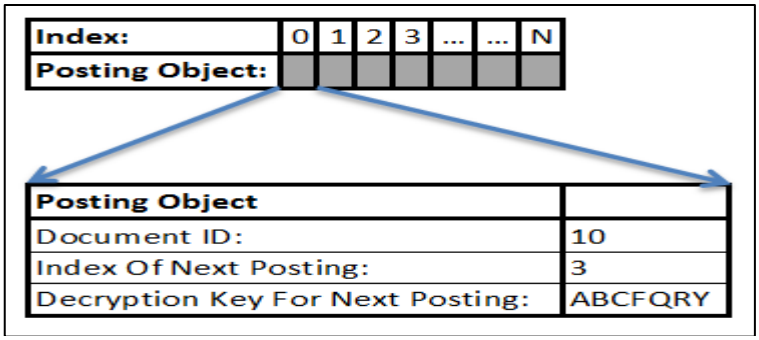

Figure 12: Postings Array Node (Including Decryption Key Storage).

As regards encryption algorithms, the SSE literature states that any standardised secure symmetric algorithm can be utilised for Posting/Posting List Encryption & Document Encryption (*For Example: AES, Triple DES*). SSE requires the use of three encryption keys for SSE Inverted Index Construction and Querying. They are one key for Lexicon Encryption/Searching (used to generate a keyed hash of each Lexicon Term/Search String), one master key used to derive encryption/decryption keys for the first Posting associated with each Lexicon Term and one key for Document Collection encryption/decryption.

2.2.3 Querying an SSE Inverted Index

There are two types of SSE Schemes: Interactive; *that is, the Client and the Server exchange numerous messages before the Server responds with a set of Search Results*, and non-Interactive; *that is, the Client issues a Search String to the Server and the Server responds immediately with a set of Search Results (Bosch et al. 2014)*. The following description of querying an SSE Inverted Index covers the latter. Given that the Lexicon of the SSE Inverted Index consists of a keyed-hash of each Term within the Document Collection, the Client is therefore required to generate a keyed-hash of their Search String in order to Query the Lexicon. The resulting Search String; *that is, an Encrypted Search String (ESS)*, is then forwarded to the Server. In addition to forwarding the ESS to the Server, the Client must also forward the decryption key necessary to decrypt the first Posting associated with the ESS. In the event that the ESS is present in the Lexicon, the first Posting associated with the ESS is retrieved. The Server then proceeds to decrypt this information revealing the ID of the first Document containing the ESS, the Index Location of the second Posting, as well as the decryption key necessary to decrypt the information stored in the second Posting. This process then repeats until all Postings associated with the ESS have been retrieved and decrypted. Following this, the associated Document IDs are then used to retrieve the actual encrypted Documents – from secondary memory – that contain the ESS. Once all Documents are retrieved, they are then forwarded to the Client; *that is, Search Results (Song et al. 2000)*. In the event that an ESS is not present in the Lexicon, this denotes that the ESS in question is not present in any Documents contained within the Collection.

2.3 Kamara & Cash SSE Implementations

Despite its efficiency, the fact remains that working implementations of SSE are few and far between. Two working implementations of SSE are (Kamara *et al.* 2012, Cash *et al.* 2013). Neither of which are available in the public domain.

2.3.1 Kamara *et al.* (2012) Implementation

The implementation of SSE developed by Kamara *et al.* (2012) is a non-interactive, single Query Term SSE protocol. When compared to the description of SSE previously, the implementation by Kamara *et al.* (2012) is almost identical with the exception the implementation supports the addition and deletion of documents from the Document Collection (and therefore the SSE Inverted Index) – The description of SSE previously assumed that the underlying Document Collection was static. The implementation also stores the entire Inverted Index; that is, Lexicon and Posting Lists, in RAM at all times. In terms of programming languages, the implementation of SSE by Kamara *et al.* (2012) was developed using Microsoft C++.NET. Any and all cryptographic functionality associated with implementation employed the use of the Microsoft CNG library of cryptographic algorithms. In relation to Data Structures, the exact Data Structure used for Lexicon Storage is not disclosed by Kamara *et al.* (2012). In the theoretical description of their scheme, Kamara *et al.* (2012) endorse the use of a 'Dictionary' Data Structure for Lexicon Storage; however the exact Data Structure used in the resulting implementation is not disclosed. Given that a number of Data Structures fall under the category of Dictionary Data Structures, we can only speculate as to the exact Data Structure used. In terms of Posting List storage, Kamara *et al.* (2012) employ the use of a single one dimensional 'Array' Data Structure (as was the case with the description of SSE provided previously). In terms of algorithms, the SSE implementation by Kamara *et al.* (2012) utilises 128 bit

AES-CBC for Posting encryption, while HMAC-SHA256 is used for keyed hashing of Lexicon Terms. The exact algorithm used for Document encryption is not disclosed.

In relation to Test Data, Kamara *et al.* (2012) tested their SSE implementation against three separate Test Data Sets: a subset of the Enron E-Mail Collection (16MB in size with approximately 1.5 million Postings), a collection of Microsoft Office Documents used by one of Microsoft's Business Groups (500MB in size with approximately 650,000 Postings), and a collection of Media Files (*For Example: MP3, WMA, JPG*) (500MB in size – number of postings not disclosed). In terms of Research Results, the work of Kamara *et al.* (2012) focussed on two separate aspects of SSE: constructing the SSE Inverted Index, and querying the SSE Inverted Index. For SSE Inverted Index construction, it should be noted that the Results presented by Kamara *et al.* (2012) only take into account the process of converting a pre-existing IR Inverted Index into an SSE Inverted Index and encrypting the associated Document Collection – the Results do not include the amount of time taken to generate the initial IR Inverted Index; nor do they take into account the time associated with transferring the SSE Inverted Index and the encrypted Document Collection from the Client to the Server. For the Enron E-Mail Test Data Set, constructing the associated SSE Inverted Index and encrypting the associated Document Collection took 52 seconds. For the Microsoft Office Document Collection Test Data Set, constructing the associated SSE Inverted Index and encrypting the associated Document Collection took approximately 33 seconds. For SSE Inverted Index searching, it should be noted that the Results presented by Kamara *et al.* (2012) only take into account the process of retrieving and decrypting matching Postings from within an SSE Inverted Index that is permanently resident in RAM – the Results do not include the amount of time taken to retrieve the SSE Inverted Index from secondary memory and loading it into RAM, nor do they include the amount of time taken to retrieve matching Documents from disk and returning them to the Client. In relation to SSE search time, it should be noted that search time is dependent on the number of matching documents associated with the search Term; *that is, the more frequent the Search Term appears in the Document Collection, the longer the associated Search operation will take*. As such, a common performance measure for the SSE Inverted Index is the amount of time taken to retrieve and decrypt the set of all Postings associated with the most commonly occurring Term within the Lexicon. In relation to the Enron E-Mail Test Data Set, retrieving the set of all Postings associated with the most commonly occurring Lexicon Term took 53 microseconds (µs), while identifying same took approximately 8 microseconds (µs) for the Microsoft Office Document Collection Test Data Set. As regards hardware, the experiments conducted by Kamara *et al.* (2012) were performed on a Windows Server 2008R2 machine with an Intel Xeon L5520 Processor (2.26GHZ).

2.3.2 Cash *et al.* (2013) Implementation

The implementation of SSE developed by Cash *et al.* (2013) is a non-interactive, multiple Query Term SSE protocol. When compared to the description of SSE provided previously, the implementation by Cash *et al.* (2013) is almost identical with the exception of the following:

- The implementation supports Conjunctive and Boolean Queries – *The description of SSE previously assumed that all Queries consisted of a single Term (Conjunctive/Boolean Queries were not discussed as they were ruled beyond the scope of this research).*
- The implementation stores the entire Inverted Index; *that is, Lexicon, Posting List and Document Collection*, in secondary memory at all times (due to the fact the implementation is designed to scale to extremely large Data Sets).
- The implementation includes a RAM resident Data Structure known as an *X-Set* that works in combination with the Inverted Index Data Structure to aid with the execution of Conjunctive/Boolean queries.

In terms of programming languages, the implementation of SSE by Cash *et al.* (2013) was developed using the C programming language. Any and all cryptographic functionality associated with implementation employed the use of the OpenSSL library of cryptographic algorithms. In relation to Data Structures, a single Data Structure known as a *T-Set* is used to store both the Lexicon and the Postings in the implementation presented by Cash *et al.* (2013). In essence, a T-Set is a modified Hash Table that can store a fixed number of values, instead of a single value (as is the case with a standard Hash Table); *that is, Key => Value 1, Value 2, …, Value N (T-Set) instead of Key => Value (Hash Table)*. When stored on disk, the T-Set is subdivided into a number of smaller Hash Tables, with the size of each individual Hash Table based on the characteristics of the underlying Operating System and storage medium. In terms of algorithms, the SSE implementation by Cash *et al.* (2013) utilises AES-FFX for Posting encryption, while AES-HMAC or AES-CMAC is used for keyed hashing of Lexicon Terms. The exact algorithm used for Document encryption is not disclosed. In relation to Test Data, Cash *et al.* (2013) tested their SSE implementation against three separate Test Data Sets: the entire Enron E-Mail Collection (1.5 million Documents consisting of 1.2 million distinct Lexicon Terms), a 100,000 record Database generated from census data, and a number of subsets of the ClueWeb09 collection of crawled web pages (the largest of which was 410GB in size (13,284,801 HTML Files) with approximately 2.7 billion

Postings). In terms of Research Results, the work of Cash *et al.* (2013) focussed on querying the SSE Inverted Index using both single Term Queries and Conjunctive/Boolean Queries. As was the case with Kamara et al. (2012), the Results presented by Cash *et al.* (2013) only take into account the process of retrieving and decrypting matching Postings from within the SSE Inverted Index itself– the Results do not include the amount of time taken to retrieve matching Documents from disk and returning said Documents to the Client. Identifying and decrypting those Postings associated with the most frequently occurring Lexicon Term for the Enron E-Mail Test Data Set (690,492 Postings) took approximately 70 seconds (approximately 100 microseconds (µs) per Postings). Unlike, Kamara *et al.* (2012), Cash *et al.* (2013) does not disclose the amount of time taken to generate an ESS. As regards hardware, the experiments conducted by Cash *et al.* (2013) were performed on an IBM Blade HS22 running a Linux operating system, with all secondary memory provided by a Storage Attached Network (SAN) device.

2.3.2.1 Critical Analysis of Existing Implementations

From the results presented by Kamara *et al.* (2012) and Cash *et al.* (2013), it is apparent that the search time associated with SSE is impressive – to the point that one could argue SSE is efficient enough to be deployed in a Cloud environment. In addition, the work of Cash *et al.* (2013) proves that SSE does indeed scale to large Data Sets whilst maintaining its search efficiency, and also has the ability to support Boolean/Conjunctive Queries in an efficient manner whilst maintaining Data/Query Privacy. Despite such impressive results, it seems that both papers focussed on the performance of a single component of SSE; *that is, searching an SSE Inverted Index*, and not SSE as a whole. Specifically, both have glossed over the topic of SSE Inverted Index Construction. Given that constructing an SSE Inverted Index is a necessary pre-requisite to searching an SSE Inverted Index; the topic deserves significantly more attention than that which it has been given in the published literature thus far. Kamara *et al.* (2012) cover the topic briefly in their work; however as indicated previously, the results presented are somewhat skewed by the fact they only include the Results of converting a pre-existing IR Inverted Index into an SSE Inverted Index – the results do not include the time taken to generate the initial IR Inverted Index. Cash *et al.* (2013) make no mention of the time taken to generate the SSE Inverted Index used in their work.

In addition to largely ignoring the process of constructing an SSE Inverted Index, both papers have also ignored the process of transferring the SSE Inverted Index and the encrypted Document Collection from the Client to the Server. As Kamara *et al.* (2012) correctly points out, the time taken to transfer both the SSE Inverted Index and the encrypted Document Collection from the Client to the Server will vary depending on the underlying system (Kamara et al. (2012) failed to cover this part of SSE for this reason); however the same can also be argued in relation to cryptographic operations (which are of course reported on in detail in both implementations). When discussing their Results in relation to searching an SSE Inverted Index, both Kamara *et al.* (2012) and Cash *et al.* (2013) readily acknowledge that their Results only cover searching the SSE Inverted Index and decrypting the Postings associated with the Lexicon Term being searched – their Results do not include the time associated with retrieving and forwarding matching Documents to the Client – another essential component of SSE. In addition to their failure to examine SSE as a whole, there was a sparsity in information relating to the test data sets and findings of both papers. In relation to Test Data, Table 1 summarises the Test Data statistics published (and not published) in both papers.

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Number of Documents In Data Set	No	Yes
Number of Terms In Data Set	No	No
Number of Unique Terms In Data Set	No	Yes (Enron Data Set Only)
Number of Postings In Data Set	Yes (Postings In Media File Data Set Not Disclosed)	Yes (Postings In Census Data Set Not Disclosed)
Number of Postings Associated With Highest Frequency Lexicon Term	No	Yes (Not Disclosed For Media File Data Set)
Size of Test Data Set	Yes	Yes (Size Of Census Data Set Not Disclosed)

Table 1: Test Data Statistics

The total number of Terms in the Data Set is relevant in that it dictates the amount of work needed to be performed during Document Tokenisation; *that is, IR Inverted Index Construction*, the number of unique Terms in the Data Set is relevant in that it dictates the number of Terms contained within the Inverted Index (both the IR Inverted Index and the SSE Inverted Index), while the number of Postings in the Data Set is relevant in that

it dictates the number of Postings contained within the Inverted Index (both the IR Inverted Index and the SSE Inverted Index). The number of Postings associated with the highest frequency Lexicon Term is relevant in that the Term in question is typically used to measure the worst case scenario of searching an SSE Inverted Index, while the size of the Test Data Set is relevant in terms of transmitting the Document Collection to the Server from the Client. As can be seen from Table 1, a number of these statistics are not disclosed (or are only partially disclosed); therefore making it difficult to give context to the associated experiment results. In relation to Inverted Index Construction statistics, Table 2 summarises the Test Data statistics published (and not published) in both papers.

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Time Taken To Generate IR Inverted Index	No	No
Size Of IR Inverted Index	No	No
Time Taken To Convert IR Inverted Index To SSE Inverted Index	Yes	No
Size of SSE Inverted Index	No	Yes
Time Taken To Encrypt Document Collection	Yes	No

Table 2: Inverted Index Construction Statistics

The time taken to generate the IR Inverted Index is significant in that the processing time is linear in the number of Terms contained within the Document Collection. The time taken to generate the SSE Inverted Index is significant in that the processing time is linear in the number of Postings contained within the IR Inverted Index, while the size of the SSE Inverted Index is relevant in terms of transmitting the SSE Inverted Index to the Server from the Client. As can be seen in Table 2, neither Kamara *et al.* (2012) or Cash *et al.* (2013) disclose any information in relation to IR Inverted Index Construction. When reporting the Results of converting their IR Inverted Index to an SSE Inverted Index, Kamara *et al.* (2012) choose to do so by charting their Results against the size of the Test Data Set (in MB). This information would be much more informative if it were charted against the number of Postings in the Test Data Set, given that the size of the underlying Data Set in no way reflects the number of unique Terms or Postings in the Data Set. *For Example: a 10MB DOCX file may contain the same Term repeated over and over again; that is, one unique Term => one Posting*. In addition, the use of the Document Collection size here is a poor choice given the fact that different file formats can contain the same number of words, but differ greatly in size (such a TXT Files and DOCX Files). In relation to Inverted Index Querying statistics, Table 3 summarises the Test Data statistics in both papers.

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Time Taken To Generate ESS	Yes	No
Time Taken To Search SSE Inverted Index For Highest Frequency Lexicon Term (Including Decryption Of Postings)	Yes	Yes

Table 3: Inverted Index Querying Statistics.

As can be seen from Table 3, both Kamara *et al.* (2012) and Cash *et al.* (2013) have disclosed the time taken to search the SSE Inverted Index and to identify and decrypt the Postings associated with the highest frequency Lexicon Term. Unfortunately Kamara *et al.* (2012) did not publish the number of Postings associated with the highest frequency Lexicon Term; instead the amount of time associated with the search was published. Without the number of Postings associated with the highest frequency Lexicon Term, it is difficult to place into context the significance of the results published. In the case of Cash *et al.* (2013), both the search time and the number of Postings associated with the highest frequency Lexicon Term were published, therefore providing readers with the ability to estimate the amount of time required to decrypt a single Posting. Here 700,000 Postings were identified and decrypted in approximately 70 seconds for the highest frequency Lexicon Term (approximately 100 microseconds per Posting). In relation to the time taken to generate an ESS, only Kamara *et al.* (2012) have published their Results for this area. While Cash *et al.* (2013) have not revealed their statistics for this part of SSE Search, it is safe to assume that the cost of producing an ESS is miniscule given that the implementation developed by Kamara *et al.* (2012) does so in 35 microseconds (µs). In relation to Test Environment statistics, Table 4 summarises the statistics published (and not published) in both papers.

Information Disclosed	**Kamara *et al.* (2012)**	**Cash *et al.* (2013)**
Operating System	Yes	Yes
Processor	Yes	No
RAM	No	No
Hard Disk Size	No	No

Table 4: Test Environment Statistics.

There is simply a lack of information disclosed in both papers regarding the underlying Test Environments (see Table 4). In an effort to determine the performance cost of preserving Data/Query privacy using SSE, Cash *et al.* (2013) opted to perform a performance comparison between their implementation of SSE and a MySQL Server comprising a plaintext database. A comparison between an equivalent plaintext Information Retrieval (IR) system would be a much more appropriate comparison to make when determining the performance cost of SSE (given the fact that plaintext searching is the universally accepted method of IR); nonetheless, the decision to perform a comparison against MySQL can be explained by the fact that their implementation of SSE is optimised for searching large Data Sets stored in secondary memory and not primary memory (as is the case with all Database Servers – including MySQL).

3. Evaluation

We have therefore identified a number of issues with the information available regarding existing implementations of SSE. In addition, we have identified that research into SSE has almost exclusively focussed on the topic of searching in SSE, while largely ignoring the topic of SSE Inverted Index Construction. Our intention is to contribute towards the areas of weakness identified. With this in mind, we have identified the following Research Questions:

- RQ1: *How Efficient Is Searchable Symmetric Encryption (SSE) When Implemented And Deployed In A Cloud Environment?*

- RQ2: *What Is The Performance Cost Of Preserving Data/Query Privacy Using Searchable Symmetric Encryption (SSE) When Compared To Plaintext Information Retrieval (IR)?*

The existing SSE literature has failed to cover the whole spectrum of activities associated with SSE (see Table 5); hence RQ1. Additionally, the existing published literature has yet to examine the usage of SSE when deployed in a Cloud computing environment. In relation to RQ2, the existing published literature has only compared the performance of SSE with a Database Server, and not a traditional plaintext IR system that utilises an Inverted Index (Cash *et al.* 2013).

Activity	**Covered In Existing Literature**
SSE Inverted Index Construction	
IR Inverted Index Generation By Client	No
SSE Inverted Index Generation By Client	Yes
Document Collection Encryption By Client	Yes
Uploading Of SSE Inverted Index To Server	No
Uploading Of Encrypted Document Collection To Server	No
SSE Inverted Index Searching	
ESS Generation By Client	Yes
Identifying And Decrypting Matching Postings	Yes
Returning Matching Documents To Client	No

Table 5: SSE Activities Covered By Existing Literature.

3.1 Experimental Setup

Both software artefacts have been developed with a view to providing answers to the Research Questions identified previously. Both artefacts are examples of personal file hosting applications. Like all file hosting applications, the objective of both the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine“ is to allow service users to store their files in the Cloud, and to access/retrieve those files as and when needed (via a web browser). In the case of the “PlainTXT Storage and Search Engine” application, users will be able to store their personal files in plaintext form, as well as having the ability to search and retrieve those files by forwarding queries to the application in plaintext form. In the case of the “CipherTXT Storage and Search Engine” application, users will be provided with the exact same functionality as the “PlainTXT Storage and Search Engine” application, with the exception that both user’s files and queries are encrypted prior to being forwarded to the application for storage/usage. Given the prototype status of both applications, a number of standard features and functionality typically associated with personal file hosting services have been classified as out of scope for the initial version of both software artefacts. Both the “PlainTXT Storage and Search Engine” and “CipherTXT Storage and Search Engine“ applications were implemented using the Java Programming Language. All Client-Side functionality associated with both applications was implemented in

the form of Java Applets, while all Server-Side functionality was implemented in the form of Java Servlets. The SSE scheme underlying the "CipherTXT Storage and Search Engine" application is (*Kamara et al., 2012)*.

The Operating System was Windows Ultimate 64-Bit SP1. The Java Development Kit (JDK) was v.8 and JRE was update 51, build 16. The Web Server (Localhost) was Apache Tomcat 7.0.56. Tests were run on an Intel Core i7 4900MQ @2.8GHz Quad Core laptop with 24GB RAM (3 X 8GB KINGSTON DDR3 @ 800MHz). The Hard Disk was a 925GB SSHD with RAID 1. All tests were conducted using the default Java Virtual Machine (JVM) - no additional runtime parameters were configured.

All experiments were performed on the '20 Newsgroups' Data Set (Rennie, 2008). In its original form, the '20 Newsgroups' Data Set consists of 18,828 files, subdivided into 20 folders. Initially, each file in the Data Set has a numeric file name between 4 and 6 digits in length with no file extension. Prior to being used in the experiments, we first attempted to move all files in the Data Set into a single folder; however at this point we noted that the names of all files in the Data Set are not unique (the contents of each file are unique however (Rennie 2008)). In an effort to avoid duplicate file names, we randomly assigned an 8 digit numeric name to each file in the Data Set. We also appended the TXT file extension to each file in the Data Set. As part of Testing, we tested each aspect of SSE with Data Sets that increased in size by an order of magnitude. As such, it was necessary to derive smaller subsets from the full '20 Newsgroups' Test Data Set. In total, 5 subsets were derived (DS1 – DS5). The details associated with each subset – and the full Data Set (DS6) – can be seen in table 6. We present the results associated with SSE Inverted Index Construction, SSE Inverted Index Searching and the comparison of SSE and plaintext Information Retrieval (IR). All results represent average values obtained over ten executions of each experiment.

Data Set Name	DS1	DS2	DS3	DS4	DS5	DS6
# of Docs	1	10	100	1,000	10,000	18,828
# of Terms	320	2,612	33,611	281,363	2,738,580	5,130,520
# of Unique Terms	206	1,297	10,996	52,134	258,463	377,880
# of Postings In Data Set	206	1,650	19,838	168,768	1,672,576	3,138,449
# of Postings Associated With Highest Frequency Lexicon Term	1 *All Terms*	10 *And*	100 *Subject:*	1,000 *From:*	10,000 *Subject:*	18,828 *Subject:*
Size	1.9KB	16.1 KB	215KB	1.7MB	17.3MB	32.3MB

Table 6: Test Data Set Statistics.

3.2 SSE Inverted Index Construction

Figure 13 denotes the Experimental Results associated with generating a plaintext Information Retrieval (IR) Inverted Index for each Test Data Set outlined previously. Figure 13 compares the time taken to generate the IR Inverted Index against the number of Terms in the Document Collection; *that is, the Test Data Set*, from which the IR Inverted Index is being generated.

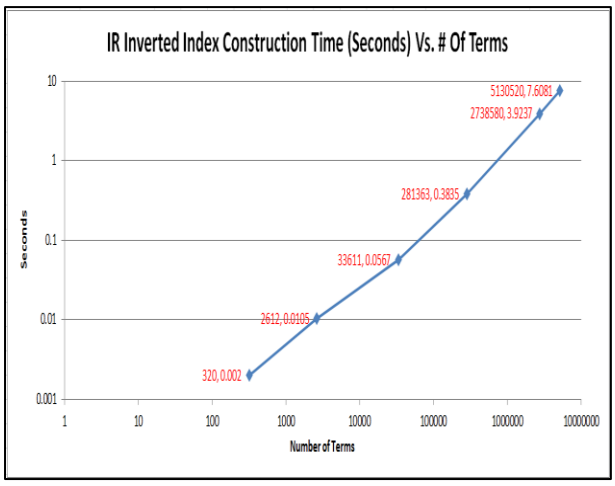

Figure 13: Information Retrieval (IR) Inverted Index Construction Time vs. Number of Terms in Collection.

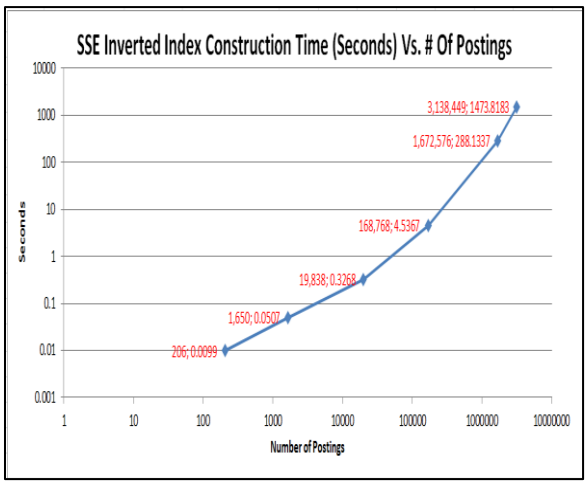

Figure 14: SSE Inverted Index Construction Time vs. No of Postings in IR Inverted Index.

As can be seen in Figure 13, the time associated with constructing an IR Inverted Index appears to increase linearly as the number of Terms in the underlying Document Collection increases. In relation to Test Data, an IR Inverted Index was generated for Test Data Set 6 (approximately 5 million Terms) in approximately 7.6 seconds. The Results shown in Figure 13 were obtained by executing `IR_Inverted_Index_Construction_Time.java` on each Data Set outlined previously.

Figure 14 denotes the Experimental Results associated with converting a plaintext Information Retrieval (IR) Inverted Index into an SSE Inverted Index for each Data Set outlined previously. Figure 14 compares the time taken to generate the SSE Inverted Index against the number of Postings in the IR Inverted Index from which the SSE Inverted Index is generated. For the first four Test Data Sets (DS1 – DS4), the time associated with constructing an SSE Inverted Index appears to increase linearly as the number of Postings in the underlying IR Inverted Index increases; however the time taken to generate an SSE Inverted Index for DS5 and DS6 increases dramatically (when compared to the number of Postings in the underlying IR Inverted Index). In relation to Test Data Sets, an SSE Inverted Index was generated for Test Data Set 4 (281,363 Postings – approximately 3.2 Postings per Lexicon Term) in 1.5 seconds. For Test Data Set 5 (1,672,576 Postings – approximately 6.5 Postings per Lexicon Term), an SSE Inverted Index was generated in 4 minutes 48 seconds. For Test Data Set 6 (3,138,449 Postings – approximately 8.3 Postings per Lexicon Term), an SSE Inverted Index was generated in 24 minutes 34 seconds.

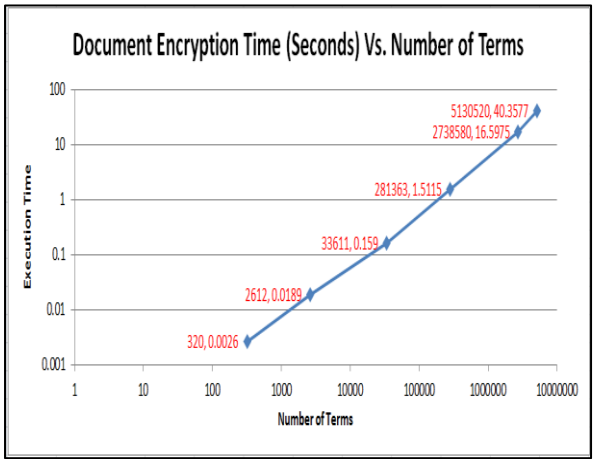

Figure 15: Document Collection Encryption Time vs. Number of Terms in Collection.

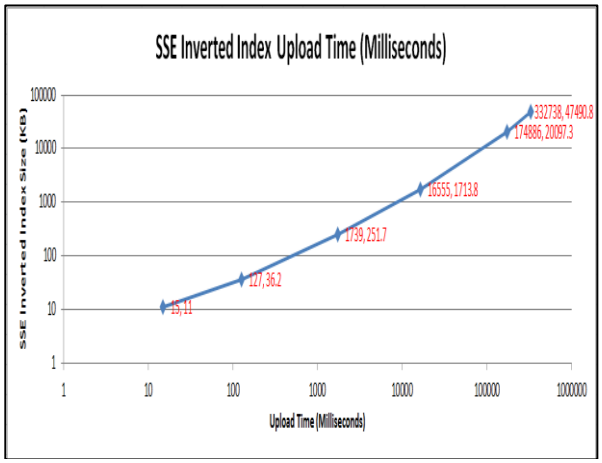

Figure 16: SSE Inverted Index Upload Time vs. Size of SSE Inverted Index

Figure 15 denotes the Experimental Results associated with encrypting the Document Collections comprising each of the Test Data Sets. Figure 15 compares the time taken to encrypt each Document Collection against the total number of Terms contained within each Document Collection. As can be seen in Figure 15, the time associated with encrypting the Document Collection appears to increase linearly as the number of Terms in the underlying Document Collection increases. In relation to Test Data Sets, the Document Collection associated with Test Data Set 6 was encrypted in 40 seconds. Figure 16 denotes the Experimental Results associated with uploading an SSE Inverted Index (generated from each Test Data Set) to the Server. Figure 16 compares the time taken to upload the SSE Inverted Index to the Server against the size of the SSE Inverted Index. As can be seen in Figure 16, the time associated with uploading the SSE Inverted Index to the Server appears to increase linearly as the size of the SSE Inverted Index increases. In relation to Test Data, the SSE Inverted Index associated with Test Data Set 6 (325MB) was uploaded to the Server in 47.5 seconds. The reader should be aware that the Experimental Results presented in Figure 16 includes the time taken to upload the SSE Inverted Index to the Server, as well as the time taken to serialise the SSE Inverted Index to disk (once the SSE Inverted Index has been received by the Server).

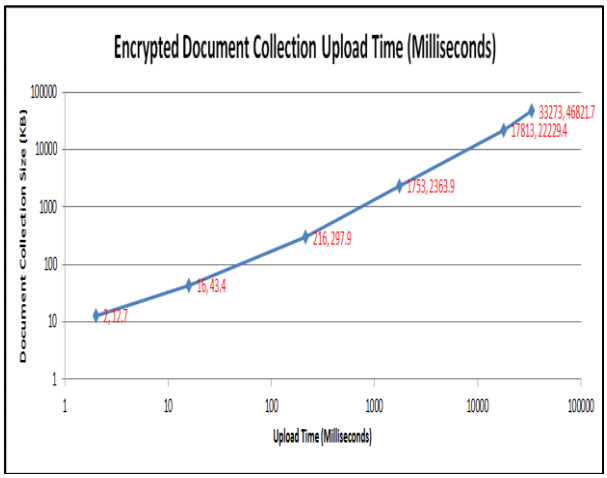

Figure 17: Encrypted Document Collection Upload Time vs. Encrypted Document Collection Size.

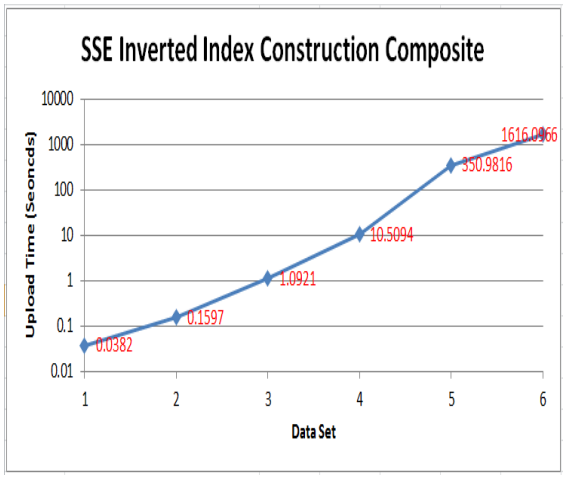

Figure 18: SSE Inverted Index Construction Composite.

Figure 17 denotes the Experimental Results associated with uploading an encrypted Document Collection (generated from each Test Data Set) to the Server. Figure 17 compares the time taken to upload the encrypted Document Collection to the Server against the size of the encrypted Document Collection. As can be seen in Figure 17, the time associated with uploading the encrypted Document Collection to the Server appears to increase linearly as the size of the encrypted Document Collection increases. In relation to Test Data, the encrypted Document Collection associated with Test Data Set 6 (32.5MB) was uploaded to the Server in 46.8 seconds. The reader should be aware that the Experimental Results presented in Figure 17 include the time taken to upload the encrypted Document Collection to the Server, as well as the time taken to store the encrypted Document Collection on disk (once the encrypted Document Collection has been received by the Server). Figure 18 denotes the total time taken to create an IR Inverted Index, convert it to an SSE Inverted Index, encrypt the associated Document Collection and upload both the SSE Inverted Index and the encrypted Document Collection to the Server for each Test Data Set outlined previously. In relation to Test Data, the whole process of constructing an SSE Inverted Index and uploading all associated data to the Server took 10.5 seconds for Test Data Set 4. To carry out the same work on Test Data Set 5 took 5 minutes 50 seconds, while carrying out the same work on Test Data Set 6 took 26 minutes 56 seconds.

3.3 SSE Inverted Index Querying

Figure 19 denotes the Experimental Results associated with generating Encrypted Search Strings (ESS) for SSE. For each Lexicon Term within the Test Data Sets outlined previously, an ESS; *that is, a keyed hash*, was generated.

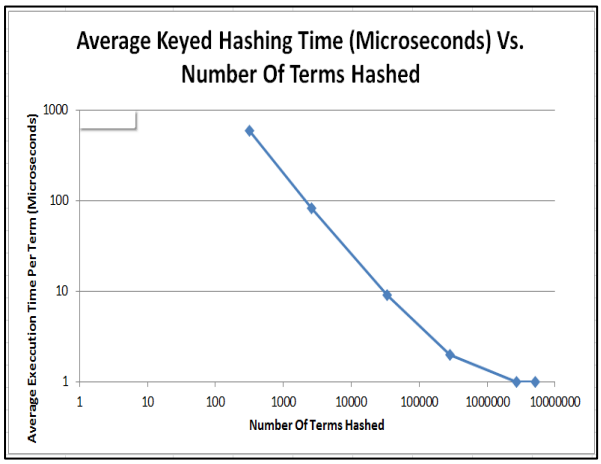

Figure 19: Encrypted Search String (EES) Generation Time vs. Number of Terms in Document Collection

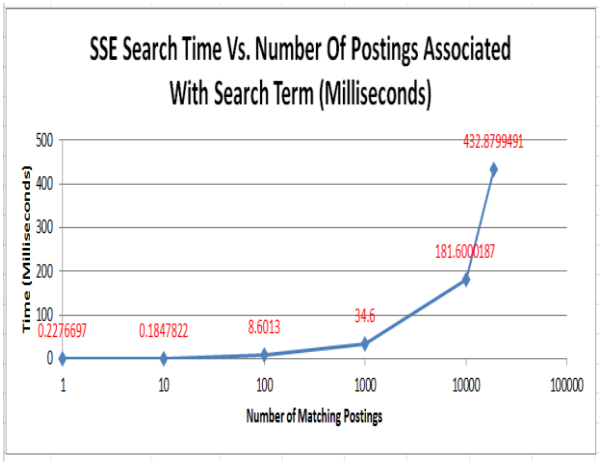

Figure 20: SSE Search Time vs. Number of Matching Postings in SSE Inverted Index

As can be seen in Figure 19, the time taken to generate an ESS is by no means constant. The Experimental Results appear to show that the more ESS that are generated, the faster the execution time of the underlying `Keyed_Hash()` Method. Figure 20 denotes the Experimental Results associated with searching an SSE Inverted Index and identifying (and decrypting) the Postings associated with the most frequently occurring Lexicon Term within the underlying Document Collection. Figure 20 compares the time taken to search the SSE Inverted Index against the number of Postings associated with the most frequently occurring Lexicon Term within the underlying Document Collection. In relation to Test Data, the SSE Inverted Index associated with Test Data Set 6 was searched and all Postings associated with the most frequently occurring Lexicon Term (18,828 Postings) were identified in 432 milliseconds.

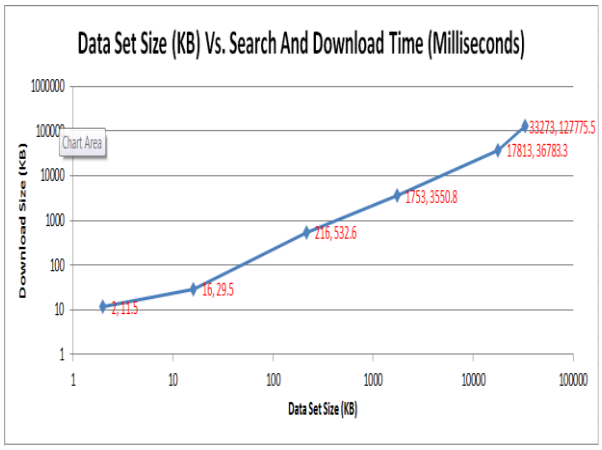

Figure 21: Data Set Size vs. Search and Download Time.

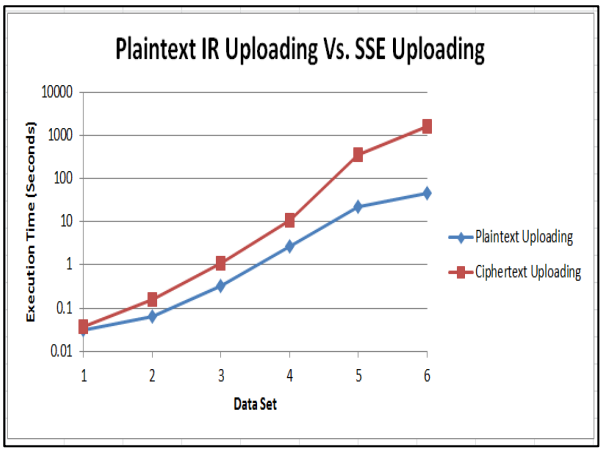

Figure 22: Plaintext IR Uploading vs. SSE Uploading.

Figure 21 denotes the Experimental Results associated with searching an SSE Inverted Index for the most frequently occurring Lexicon Term within the underlying Document Collection and returning all matching Documents to the Client. Figure 21 compares the time taken to search the SSE Inverted Index and return all matching Documents against the size of the Document Collection returned. The reader should be aware that the Experimental Results presented in Figure 21 also include the time taken to encapsulate the set of all matching Documents within a ZIP File, which is then returned to the Client. In relation to Test Data, the set of matching Document associated with the most frequently occurring Lexicon Term contained within Test Data Set 6 was searched and all Documents returned to the Client (32.5 MB) in 2 minutes 7 seconds.

3.4 Performance of SSE vs. Plaintext Information Retrieval (IR)

Figure 22 denotes the results associated with the comparison of traditional plaintext Information Retrieval (IR) uploading and SSE uploading. Those values associated with IR uploading in Figure 22 represent the time taken to upload the Document Collection associated with each Test Data Set from the Client machine to the Server. Those values associated with SSE uploading in Figure 22 represent the time taken to generate the SSE Inverted Index, encrypt the associated Document Collection, and uploading both the Inverted Index and encrypted Document Collection to the Server. It is obvious that the amount of time necessary for SSE uploading increases in a non-linear manner when compared to the amount of time necessary for plaintext IR uploading.

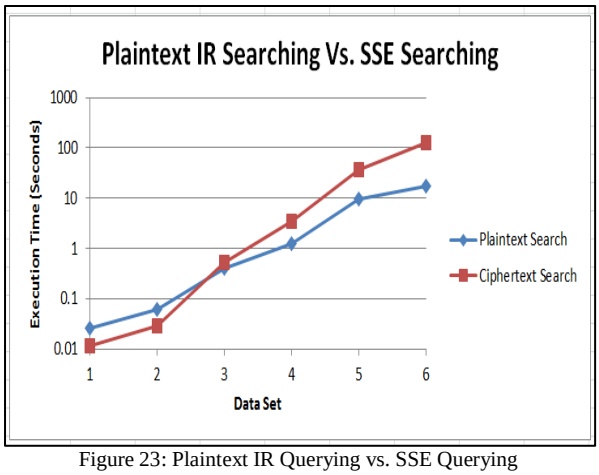

Figure 23: Plaintext IR Querying vs. SSE Querying

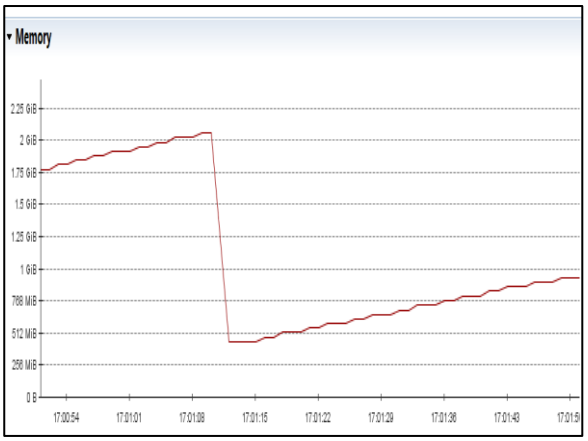

Figure 24: Java Heap Memory Usage and Garbage Collection Statistics for SSE Inverted Index Construction

Figure 23 denotes the comparison of traditional plaintext Information Retrieval (IR) querying and SSE querying. The Experimental Results presented in Figure 23 consist of the time taken to identify the set of all Postings associated with the most frequently occurring Lexicon Term in the underlying Document Collection, and encapsulating the set of all matching Document within a ZIP File which is then returned to the Client. It is obvious from Figure 23 that the amount of time necessary for SSE querying increases in a non-linear manner when compared to the amount of time necessary for plaintext IR querying.

3.5 Summary

In relation to searching an SSE Inverted Index, the results provide additional proof of the efficiency of SSE when implemented in software. The implementation of SSE developed as part of this research was able to identify and decrypt a single Posting associated with a given Lexicon Term in approximately 22 microseconds (µs). This performance is comparable with the implementations of SSE developed by Kamara *et al.* (2012) which was 7.3 Microseconds (µs) per Posting and Cash *et al.* (2013) which was 100 Microseconds (µs) per Posting. Regarding the efficiency of constructing an SSE Inverted Index, the results are somewhat inconclusive. Given the five steps involved in constructing an SSE Inverted Index, each step in the implementation of SSE produced as part of this research performed as expected with the exception of the second step: *Converting an IR Inverted Index to an SSE Inverted Index*. For Test Data Set 1 (DS1) through Test Data Set 4 (DS4), an SSE Inverted Index was generated from an existing IR Inverted Index in a time linear to the number of Postings stored in the IR Inverted Index; however, for DS5 and DS6, this apparent linear performance decreased dramatically. This decrease in performance could be attributed to a combination of one or more of the following: 1) The Java Virtual Machines (JVM) Garbage Collection functionality, 2) Insufficient Java Heap memory, 3) The use of `String` Objects in the `Encrypted_Array_Node` Class, 4) The size of the SSE Inverted Index, and 5) The requirement of the `HashMap iterator()` Method to store an additional copy of the IR Inverted Index `HashMap` on the Java Heap while the SSE Inverted Index is being constructed. Regarding the first and second point, we dynamically analysed the 'CipherTXT Storage and Search Engine' application using both the Java Mission Control and Java Flight Recorder applications. In both cases, the JVM Garbage Collector was extremely active (eradicating up to 2GB of Objects from the Java Heap on a regular basis) (see Figure 24).

Regarding points one, two and three, it is evident that a significant number of Objects are being stored on the Java Heap as the implementation of SSE converts the IR Inverted Index to an SSE Inverted Index. One possible explanation for this is the use of `String` Objects in the `Encrypted_Array_Node` Class. `String` Objects are used in the `Encrypted_Array_Node` Class to store encrypted Document IDs, encrypted Indexes of subsequent Postings, as well as keys required to encrypt/decrypt subsequent Postings. Given that `String` is a form of Object - and not a primitive data type – all `String` Objects are therefore stored in the Heap area of the Java Virtual Machines (JVM) memory. Regarding points one, two and four, the SSE Inverted Index associated with DS5 was 171MB in size, while the SSE Inverted Index associated with DS6 was 325MB in size. When compared to their plaintext equivalent, the DS5 IR Inverted Index is 25MB in size (146MB smaller than its SSE counterpart), while the DS6 IR Inverted Index is 42.8MB (282.2MB smaller than its SSE counterpart). Evidently the SSE Inverted Index associated with both DS5 and DS6 occupy a significant amount of memory. The presence of such large Objects in the Java Heap obviously reduces the amount of space available for subsequent Objects; therefore increasing the frequency of Garbage Collection.

Regarding points one, two and five, the `iterator()` method of the `HashMap` Class may also be a factor in the performance degradation associated with DS5 and DS6. As part of the process of converting the IR Inverted Index to an SSE Inverted Index, the IR Inverted Index must first be loaded into the Java Heap, with each entry in the IR Inverted Index then being examined and subsequently added to the SSE Inverted Index. In order to examine each entry in the IR Inverted Index, the `iterator()` method must be executed on the `HashMap` Object underlying the IR Inverted Index (the `HashMap` Class does not support iteration in any other way). In order to operate, the `iterator()` method create must first create an exact replica of the IR Inverted Index `HashMap` on the Heap (that supports iteration); therefore doubling the amount of Heap space associated with the IR Inverted Index. The IR Inverted Index associated with DS5 is 25MB in size (increasing to 50MB during SSE Inverted Index Construction as a result of using the `iterator()` method), while the IR Inverted Index associated with DS6 is 42.8MB (increasing to 85.6MB during SSE Inverted Index Construction as a result of using the `iterator()` method).

We identified two other potential causes of the performance degradation associated with DS5 and DS6. These were Hash collisions occurring as a result of inserting keys into the SSE inverted index `HashMap` Object, and the natural performance degradation associated with an ever expanding `HashMap` object. Regarding Hash Collisions in a `HashMap`, the location of an Object within a `HashMap` is determined by the value resulting from executing the `hashCode()` method associated with the Object being inserted into the `HashMap`. In the event that two Objects produce the same `hashCode()` value, the `HashMap` Class must then execute the `compare()` method associated with both Objects to determine whether or not both Objects are in fact equivalent to each other. Hash Collisions should not be an issue as the `hashCode()` method associated with the `String` Class produces a 32 bit hash value (approximately 4.3 billion different Hash Values); therefore making Hash Collisions highly unlikely for data sets the size of DS5 and DS6. Regarding the natural performance degradation associated with an ever expanding `HashMap`, we noted that a Java `HashMap` Object must be created with a specified initial capacity; *that is, number of expected entries*, and a specified expected load; *that is, the percentage of the initial capacity that must be used before the capacity of the `HashMap` is increased*. In the event that that the load specified for the `HashMap` is exceeded, a new `HashMap` Object must then be constructed (this is done automatically by the `HashMap` Class). The process of constructing a new `HashMap` Object requires that each entry in the existing `HashMap` Object be retrieved, re-hashed, and inserted into the new – larger – `HashMap` Object. This should not be an issue that as we took the initial capacity and load factor of the SSE `HashMap` into consideration and constructed the `HashMap` in a manner that does not require the `HashMap` to be expanded. Regarding results relating to upload speeds and download speeds, a localhost web server was used during testing; as such, the time associated with uploading and downloading data may appear significantly faster than those which are achievable using a live system.

4. Conclusion

Given the similarity between Searchable Symmetric Encryption (SSE) and plaintext Information Retrieval (IR), it is inevitable that comparisons will be made between the two. While having a number of goals and functions in common, the fact remains that the primary goal of SSE is to provide Data and Query Privacy. Given this – as well as the fact that SSE operates in a manner that differs greatly from plaintext IR, SSE should be viewed as a separate paradigm in the context of Information Retrieval, and not an extension of plaintext IR. In order to provide Data and Query Privacy, SSE requires a significant amount of additional processing time to carry out a task when compared to the processing time associated with carrying out the same task using plaintext IR. In terms of the performance overhead of using SSE, the Research Results show that little or no correlation exists between the time associated with carrying out a task using plaintext IR, and carrying out the same task using SSE. In general, the Research Results have shown that the time taken to carry out a task using SSE is greater than the time taken to carry out the same task using plaintext IR; nonetheless, this is to be expected given that the process of uploading a Document Collection to the Server using SSE requires the Client to first generate an SSE Inverted Index, encrypt the underlying Document Collection and then upload both to the Server, as well as the need for the Sever to decrypt Postings as part of SSE querying.

The results show that carrying out a task using SSE is directly proportional to the amount of information involved. In the case of constructing an IR Inverted Index, the results show that the time taken to generate an IR Inverted Index is directly proportional to the number of Terms contained in the underlying Document Collection. Converting the same IR Inverted Index to an SSE Inverted Index is directly proportional to the number of Postings contained within the IR Inverted Index, while the time taken to encrypt the underlying Document Collection is directly proportional to the number of Terms contained within the Document Collection. In relation to searching in SSE, the time taken to identify and decrypt the set of Postings associated

with a given Lexicon Term is directly proportional to the number of Postings. Regarding the question of whether or not SSE is efficient enough to be deployed in a Cloud environment, the answer is context dependant. If deployed in an environment whereby Search Results only have to be returned to the user in small quantities (such as an Internet Search Engine (*For Example: ten results at a time*)), then SSE would be more than efficient, irrespective of the size of the underlying Data Set (due to the fact that only a small number of Postings would need to be decrypted at a given time). If deployed in an environment whereby all results must be returned at once (as was the case with the implementation of SSE developed as part of this research, as well as the implementations developed by Kamara *et al.* (2012) and Cash *et al.* (2013), SSE would only be suitable for small and medium sized Data Sets. When applied to large Data Sets, SSE querying can become inefficient as its search time is directly proportional to the number of matching Postings (which is likely to be significant for large Data Sets). Regarding the possible commercialisation of SSE, the success of such a product would undoubtedly hinge on the knowledge of those people using the product. Users of such a product would need to be aware that SSE provides Data/Query Privacy in exchange for the efficiency associated with plaintext IR, and that an SSE Inverted Index – while slow to construct for large Data Sets – is designed to achieve efficient search speeds whilst maintaining Data Privacy.

References

Bosch, C., Hartel, P., Jonker, W. and Peter, A. (2014) *A Survey of Provably Secure Searchable Encryption*. http://eprints.eemcs.utwente.nl/24788/01/a18-bosch.pdf

Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M. C. and Steiner, M. (2013) *Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries*, Crypto 2013, Part 1, LNCS 8042, pp: 353-73

Cash, D., Grubbs, P., Perry, J. and Ristenpart, T. (2015) *Leakage-Abuse Attacks Against Searchable Encryption* in Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS'15, (New York, NY, USA), pp. 668–679, ACM, 2015.

Chang, Y.-C. and Mitzenmacher, M. (2005) *Privacy Preserving Keyword Searches on Remote Encrypted Data*. http://www.eecs.harvard.edu/~michaelm/postscripts/acns2005.pdf

Chase, M. and Kamara, S. (2010) *Structured Encryption and Controlled Disclosure* http://eprint.iacr.org/2011/010.pdf

Curtmola, R., Garay, J., Kamara, S. and Ostrovsky, R. (2006) *Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions* http://eprint.iacr.org/2006/210.pdf

Eurostat (2014) *Cloud computing - statistics on the use by enterprises* http://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises

Gentry, C. (2009) *A Fully Homomorphic Encryption Scheme*, unpublished thesis (PhD), Stanford University.

Gentry, C., Halvei, S. and Smart, N. P. (2015) *Homomorphic Evaluation of the AES Circuit (Updated Implementation)* https://eprint.iacr.org/2012/099.pdf

Goh, E. (2003) *Secure Indexes* http://crypto.stanford.edu/~eujin/papers/secureindex/secureindex.pdf

Goldreich, O. and Ostrovsky, R. (1992) *Software Protection and Simulation on Oblivious RAMs*

Hashizume, K., Rosado, D. G., Fernández-Medina, E. and Fernandez, E. B. (2013) *An analysis of security issues for cloud computing* http://www.jisajournal.com/content/pdf/1869-0238-4-5.pdf

He, K., Chen, J., Du, R., Wu, Q., Xue, G., Zhang, X. (2016). *DeyPoS: Deduplicatable Dynamic Proof of Storage for Multi-User Environments*. IEEE Transactions on Computers. 2016. Vol. 66, No. 1, pp: 24-48, DOI: 10.1109/TC.2016.2560812

ICO (2015) *Monetary Penalty Notice: Staysure.co.uk Limited* https://ico.org.uk/media/action-weve-taken/mpns/1043368/staysure-monetary-penalty-notice.pdf

Kamara, S. (2013) *How To Search On Encrypted Data* http://research.microsoft.com/en-us/um/people/senyk/slides/encryptedsearch-full.pdf

Kamara, S., Papamanthou, C. and Roeder, T. (2012) *Dynamic Searchable Symmetric Encryption* Proceedings of the 2012 ACM conference on Computer and communications security, pp: 965-976, https://eprint.iacr.org/2012/530.pdf

Levick (2015) *DATA SECURITY & PRIVACY* http://levick.com/experience/specialty/data-security-privacy

Luenberger, D. G. (2006) 'Information Science' in, Princeton, New Jersey: Princeton University Press, 284-300.

Manning, C. D., Raghavan, P. and , S., H. (2008) *Introduction to Information Retrieval*, Cambridge, England: Cambridge University Press.

Mather, T., Kumaraswamy, S. and Latif, S. (2009) *Cloud Security and Privacy*, California: O'Reilly.

Nguyen, M., Chau, N., Jung, S. and Jung, S. (2014) *A Demonstration of Malicious Insider Attacks inside Cloud IaaS Vendor* http://www.ijiet.org/papers/455-F028.pdf

Rennie, J. (2008) *The 20 Newsgroups Data Set* http://qwone.com/~jason/20Newsgroups/

Song, D. X., Wagner, D. and Perrig, A. (2000) 'Practical Techniques For Searches On Encrypted Data', in Titsworth, F. M., ed., *IEEE Symposium on Security and Privacy, 2000*, Berkeley, California, 14-17 May 2000, Washington, D.C.: IEEE Computer Society, 44-55.

Stallings, W. (2014) *Cryptography and Network Security: Principles And Practices*, New Jersey: Pearson Education.

Stefanov, E., Papamanthou, C. and Shi, E. (2013) *Practical Dynamic Searchable Encryption with Small Leakage, IACR Cryptology ePrint Archive, pp: 832* https://eprint.iacr.org/2013/832.pdf

Uchide, Y., Kunihiro, N. (2016) Searchable symmetric encryption capable of searching for an arbitrary string, Security and Communication Networks, Wiley, http://dx.doi.org/10.1002/sec.1437

Wang, B., Song, W., Lou, W. and Hou, Y. (2015) *Inverted index based multi-keyword public-key searchable encryption with strong privacy guarantee,* in Proceedings of the 2015 IEEE INFOCOM International Conference on Computer Communications, (Hong Kong), April 31-May 1 2015, pp. 2092-21110

He, K., Chen, J., Du, R., Wu, Q., Xue, G., Zhang, X. (2016). *DeyPoS: Deduplicatable Dynamic Proof of Storage for Multi-User Environments*. IEEE Transactions on Computers. 2016. Vol. 66, No. 1, pp: 24-48, DOI: 10.1109/TC.2016.2560812